LIBRARY OF THE
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

510.84
Il6r
no. 649-654
cop. 2

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

SEP 27 RECD

L161—O-1096

Digitized by the Internet Archive
in 2013

http://archive.org/details/whatdoweneedinpr652gear

370.6
I46r UIUCDCS-R-74-652
no.652

Math

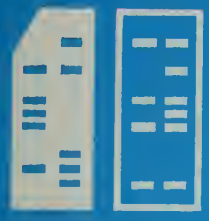
COO-2383-0009

What Do We Need In Programming Languages for Mathematical Software?

by

C. W. Gear

May, 1974



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

THE LIBRARY OF THE
JUL 18 1974
UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

UIUCDCS-R-74-652 COO-2383-0009

What Do We Need In Programming Languages For Mathematical Software?

by

C. W. Gear

May, 1974

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

*Limited Distribution. Submitted to ACM Transactions on Mathematical Software.

*Supported in part by the Atomic Energy Commission under grant US AEC AT(11-1)2383.

What Do We Need In Programming Languages for Mathematical Software?

C. W. Gear
Dept. of Computer Science
University of Illinois at Urbana-Champaign

Abstract

Existing, widely used languages are reasonably satisfactory for writing numerical subroutines. However, future mathematical software development is likely to require features not currently found in any widely used language. Those features will be needed so that the program author can specify, at a meta-language level, the way in which his program will be tailored for a particular class of problems and particular hardware characteristics. It is suggested that the most fruitful way of doing this is via a preprocessor with some macro features that produces a widely used language such as Fortran as its object. Some specific suggestions are made based on our initial experience with one such preprocessor.

1. Introduction

This paper is concerned with the language environment needed for the production of good numerical software packages. By numerical, I mean software for a problem which ultimately is solved numerically, although there may be intermediate non-numerical steps. By packages, I mean a set of programs more flexible than a fixed set of subroutines. Thus a package could be as complex as a complete problem-oriented processor for network analysis designed for non-programmer use, or as simple as a programmer callable subsystem which chooses one of several eigenvalue subroutines to handle a particular problem.

It is important that we look beyond the simple subroutine to the package, because far more computer time is spent in the numerical portion of packages than in simple library subroutines. All subroutines are important for "one-shot" runs, while shorter subroutines for common functions (SIN, etc.) are important to all programmers, but large jobs frequently have special requirements (for example, ones imposed by their data structure) that make it impossible to use standard library subroutines directly. Many large jobs are codes written for particular applications, e.g., reactor design, but a growing set of computer users are the users of problem-oriented processors. The spread of these processors makes computation available to non-programmers, and they may use a major percentage of computational resources eventually.

Until recently, all such packages were produced by application specialists (e.g. engineers), and the numerical code required was either obtained by modification of a standard library subroutine or was generated by coding a well known algorithm ("if its integration, it must need Runga

Kutta"). This in spite of the fact that the number crunching part of the package was probably the most time consuming and the least reliable part! (Recently, numerical specialists have become directly involved in some areas via efforts such as EISPACK.) If the mathematical software specialist is to have a significant impact on the efficient use of computers, he has to move as far beyond the writing of a subroutine as that step is beyond the design of an algorithm.

There are three approaches by which the numerical programmer can improve the application of numerical techniques to large problems:

(1) He can become involved in an actual problem. This was the earliest form of involvement, but has not generally led to code that can be used in a wide class of problems.

(2) He can write general purpose subroutines for distribution. This approach has flourished for nearly two decades (and we should note that it has only been possible because of "standard" languages). This approach has been successful for users who are mathematically and computationally sophisticated. Such users can choose between available subroutines (or algorithms) with discrimination, and are capable of making the necessary program adaptations to different environments, both in the machine and in the application.

(3) He can write application oriented packages which either solve the whole problem or can be interfaced

to packages which handle other parts of the problem. I favor this approach for several reasons. One is that often the specific form of the numerical problem is apparent only to someone involved in problem formulation. Frequently in the past this form has been modified by an application specialist to fit it to an available subroutine when it would have been simpler, more stable, and more efficient to handle the original problem directly. A second reason is that modification of numerical subroutines by anybody not extremely familiar with both the underlying theory and the reasons behind the implementation techniques used is very likely to cause errors that will be blamed on the algorithm rather than on that instance of implementation. Thus, when a subroutine fails to find the roots of a very non-linear system of equations, or takes an excessively long time to do so, it could be an inherent difficulty in the algorithm, or it could be caused by a "minor" change to the code, and it is doubtful that anybody except for the original author or someone equally motivated could diagnose the trouble.

It is presumptuous of us to think that, while the application specialist is not competent to work with our subroutines, we are competent to solve all the problems in his specialty. Therefore, we must plan packages that can interface directly to the mathematically sophisticated

user (one who can describe his program in terms of macroscopic mathematical operations such as eigenvalue extraction) but that can also interface to other programs which themselves will interface to the non-mathematical user in a problem oriented language. We must provide the software tools for solving the mathematical part of a problem in a form that can be adapted to a variety of situations, and yet can be guaranteed to remain at least reasonably free of error after such changes. Adaptions may be required in order to be acceptable to a particular computer or to interface to a particular problem solving system. This means that we must retain complete control over the mathematical or numerical core of our program, and be able to specify precisely the forms it can take. To do this while providing for change, we must build a protective layer around the core of our program, and try to limit changes to the more flexible and less demanding outer layers.

We have been moderately content with existing languages for writing numerical subroutines, but they are not adequate for packages. In this paper I wish to examine what we need in a language, compare that to what exists, and to consider the implementation of such a language as a preprocessor. A preprocessor is chosen not to save development time (although it does that very well) but to meet several of the requirements I claim are needed in a language.

The approach we will discuss has been forced on us as a result of the preparation of large program packages for general networks. Early work was done in a mixture of Fortran (for numeric code) and assembly language (for non-numeric code). The latter was used because of the poor object speed of the old PL/I. (360 assembly language is only a little less portable than PL/I.) The assembly code met the usual disasters - programmers leaving,

poorly defined and overly complex data interfaces, and virtual lack of portability to other 360s because of many local macro and system features that were used by the programmers.

2. Requirements

Hoare (1973) lists five objective criteria for languages and their translators as "Simplicity, Security, Fast Translation, Efficient Object Code, and Readability." (He also points out that the only widely used languages that meet these criteria are Fortran, Lisp, Algol 60, and a few of their derivatives.) Note that Hoare does not judge a language independently of the efficiency of its translator. Neither can the mathematical software writer ignore questions of efficiency. I feel that our objective is *to write a specification which can be used to generate an efficient object code for solving prescribed subclasses of problems on a prescribed machine*. The subclass of problems and the machine should both be some sort of parameters to the specification. The most obvious and trivial way in which this can be done is by existing subroutine parameters. For example, the value of one parameter could indicate which subclass of, say, ordinary differential equations, are to be handled, such as Stiff, Non-Stiff, Linear, Non-Linear, Don't-Know, while another might indicate which machine, or its word length. However, unnecessary code will have to be loaded and executed, so this will not be the most efficient way and we must look for language facilities that will permit better techniques. For example, if a truly general purpose ordinary differential equations integrator is to be written, (it hasn't yet) it will be a large source program. However, in any use, only part of the program will be needed because only a subclass of

all problems are to be solved. Some of the selection necessary can occur prior to execution time by user request. The rest cannot be made until execution time because it will depend on problem data yet to be generated - for example, data concerning the sparseness of matrices of partial derivatives which may have to be generated symbolically or numerically.

It is important that the mathematical program author retain firm control over the selection of code used to solve a particular problem, so it is important that he have a language in which he can specify how that selection is to be done. This language must allow for the generation of object code that is not cluttered nor slowed by unused code or storage areas allocated to it. It is also important that the mathematical program author stay as "close" to the floating point hardware as possible. He must be able to make environment inquiries that not only determine the range of integer and floating point numbers, but also determine the minimum precision of floating point, and whether any special cases of bad rounding occur, e.g., $a > b$ and $c \geq o$ but $ac < ab$. (Of course, the hardware should not do this, but nobody is perfect, particular not some computer manufacturers.) He must also be able to stay "close" to the machine in the sense that he must know what forms of code will be efficient for particular hardware as seen through the translator so that he can write code that will not need to be modified by a user for reasons of efficiency. Since the precise nature of the code may depend on the machine, there may have to be some selection process.

What we need is a language (and related processors) that allows code which can be tailored to a particular problem and machine at compile, load and run time. The selection of the subroutine needed should be done by the computer (that is, by the program author) based on the information

available or derivable from the problem. Hence the language should allow the user of the code to specify the nature of his problem (which will be used in the selection process) in a natural way, and, very importantly, the outer layers of the code which interface with other programs and the user should be in such a form that they can be modified easily with a low probability of causing errors in the core code. It is foolish to expect that the author of a package can foresee all potential requirements, and he should be encouraged to plan for later modification by others - including modifications to make a program in a "standard" language run on another machine! Since change is bound to happen, the language must encourage a well structured, documented code that can be easily understood by another and that can be changed "locally". Such changes are ones that do not require a knowledge of the workings of large sections of the code in order to verify correctness, so that, if the original program had been proved correct and we knew all the valid assertions, then we could prove the correctness of the modified version locally.

Dijkstra (1972), pages 39-40, has considered this problem. He says,

> "Certainly, one of the properties of large programs is that they have to be modified in the course of their lifetime. A very common reason is that the program, although logically correct, turns out to evoke unsatisfactory computations (for instance, unsatisfactory in one or more quantitative aspects). A second reason is that, although the program is logically correct and even satisfactorily meeting the original demands, it turns out to be a perfect solution

> for not quite the right problem; one is faced with a restatement of the problem and adaptation of the program."

In considering the relationship between two versions of the same program, Dijkstra goes on to state

> "The intentions are
>
> (1) that the two versions share their respective correctness proofs as far as possible;
>
> (2) that the two versions share (mechanically) as far as possible the common (or "equal") coding;
>
> (3) that the regions affected by the modifications are already well-isolated, a condition that is not met when the transitition requires 'brain-made' modifications scattered all over the text."

The technique proposed by Dijkstra is refinement. If we view the top down programming process as a tree, only a few of the lower nodes will differ between versions. Therefore, we should structure our programs so that the interfaces between the core code and the users are confined to a few nodes grouped in as small a subtree as possible. The process of refinement used by the original programmer should be very clear from the language or documentation - and preferably from the former. If program proof is possible, the assertions in a proof should appear alongside the program, and even if a precise proof does not seem possible, all "assertions" believed to be true should be stated in the original program. These will be useful in debugging both the original version and subsequent changes. For example, if the proof depends on the fact that the matrix B is positive definite at some point

(although rounding or coding errors may have invalidated that), the statement "B SHOULD BE POSITIVE DEFINITE" should appear in the text of the code. During debugging, that property could be tested. If a user modifies the code that generates B, he can test it. In fact, the language should allow assertions to be made in the form of "ASSERT < boolean expression >." Such assertions could be compiled in for debugging runs, or left as documentation in finished code. (Just that checking is always done in the EISPACK routines when the user claims that his matrix is positive definite. See Cody (1974).)

The language must allow the reasonable representation of the type of non-numerical algorithms that are used in the initial stages of problem solution, that is, it must allow manipulation of data structures and possibly some symbolic computation. Dynamic memory allocation for data areas is essential so that secondary storage is not used unnecessarily. (A virtual memory machine would alleviate this in part, although a very large package could exhaust the name space in existing virtual memory systems if dynamic allocation was not used.) An equivalent memory allocation is needed for program areas for the same reason. It must be possible to work with several types of data (numerical, character, and pointer) and to extend those types to, for example, polynomials, multinomials, rational polynomials, or symbolic expressions. (See Hull and Hofbauer (1974) for a proposal to extend the precision of numeric data.) If an execution time package can be written to handle a class of data, it should be easy to use that through the language. (By "easy", I do not mean writing "CALL SUM(A,B,C)" to perform "A = B + C" in, say, rationals.)

We need, as everybody else does, speed of the object code, and portability of the source or possibly the object. We need them, I think, more than in many other areas because our objective is precisely to prepare and distribute efficient, reliable code. Our colleagues who specialize in languages, their translators, and systems are not usually very concerned with

execution time efficiency because most of their problems are I/O limited. For example, "--efficiency per se is not terribly important. What is important is that inefficiency becomes an excuse for poorly structured programs." - Wulf (1972). It is true that they can equally justifiably complain that we have given insufficient thought to larger aspects of problem solution in the past. The purpose of this type of study is to suggest language features which we feel are necessary and also to propose methods of implementation which will make these features available in a useable form now. This form must allow us a degree of experimentation with several variants of the features we need in order that we can come to some consensus on what should be permanently embodied in any successor to Fortran or Algol 60 that is used by the numerical community, but it must also allow us to continue to use existing software and any written in a language variant that is eventually dropped. We must be able to interface to the large body of existing code. Any language without that capability is lost. To quote Baecker (1972), "...one reason why Algol 60 has 'lost' to Fortran...is the facility for separate compilation of subroutines in Fortran." (Separate compilation implies the ability to link to a library.) In fact, we can never stop being pragmatic in our approach, as it is doubtful we could ever find the resources to start over in the preparation of systems and compilers that meet our needs precisely - and even then, they would fail to provide code usuable by the rest of the programming community who will work with existing systems. We must work with existing facilities and experiment with the new rather than looking to adopt a new, fully specified but unproven language overnight. This may seem reactionary to our colleagues, but, to use, some of their absolutes seem equally conservative. To quote Baecker again (he is proposing a change to Algol 68):

"Doubtless we have erred grievously and failed to preserve the integrity of the Report. We have also failed to preserve orthogonality completely. We pray that the clerics will forgive our offense, and dig our errors out of the mire."

Present Languages

Algol and Fortran are the obvious candidates for numerical software. The 1973 CACM index of published algorithms listed less than 5% in other than these languages. Of 76 in these languages, I would classify them as shown in Table 1. "CACM" means that the algorithm

Table 1. 1973 Published Algorithms

	Algol		Fortran	
	CACM	Other	CACM	Other
Non numerical	10	9	4	5
Numerical	8	15	21	4

appeared in CACM, "other" means it appeared in Num. Math., Comp. Journ., or other European based publication. These figures are undoubtedly biased by publication restrictions (which have generally relaxed) and by manufacturer dominance. In spite of this, I think the preference of numerical programmers for Fortran is fairly clear in America.

The preference is well founded when one examines object code efficiency. A series of tests were run on the Fortran, Algol W and PL/I compilers on the 360/75 at the University of Illinois. A variable order, variable step integrator was written in each of these languages, using the

same structure, but making use of any short-cuts available (array assignment and downward do/for loops, for example). Each program was used to solve a simple pair of differential equations, but provision was made in the calling program to request the solution of up to ten simultaneous equations, the ten being obtained by replication of the pair five times. The equations were solved using different error control requests and different numbers of equations. The execution time t and the number of steps s were measured. If the number of equations is n, the time should be given by $t = B + Cs + Dns$. Since the installation is multiprogrammed, times are somewhat random, so 15 separate tests were run for three values of s and five values of n. B, C, and D were found by a least-squares fit. D is a measure of the time taken inside do/for loops as the only difference is to increase the number of times various loops are executed. C is a measure of the overhead of miscellaneous logic (a few if-then-else constructions) and subroutine calling time. B measures uninteresting overhead and is not given. Two versions of Fortran were used. Level H is an optimizing compiler (optimizing was specified). Two versions of PL/I were used. PLIX is an optimizing compiler. It was run with and without optimizing requested. The measurements yielded the results shown in Table 2. It is very clear that Fortran is superior in speed for the numerical

Table 2. Execution Time Tests on Fortran, PL/I and Algol W

	C	D
Fortran H	1.85	.313
Fortran G	1.87	.721
PLIX Optimizing	2.31	.412
PLIX	2.20	.953
PLI	9.15	1.209
Algol W	2.35	1.681

section of a program, and it is often unreasonable to forego this speed for the sake of a more flexible language. The new PLIX optimizing compiler is satisfactory in many ways. The PL/I language provides dynamic memory allocation and an adequate set of basic data types for most use, and the new compiler produces code that is only 25% to 30% slower than that of the optimizing Fortran compiler. PL/I is slightly better than Fortran as far as structure goes, but shares many faults with Fortran - it is not easily extendable, and is less desirable than Fortran when it comes to portability and interfacing with other program packages. Many new languages exhibit the structural and extensible features we need, but, to my knowledge, all fail miserably in execution speed (e.g. PASCAL) and are decidedly less portable than Fortran.

To listen to this refrain over the virtues of Fortran, one could be reminded of the scepticism of assembly language programmers to Fortran two decades ago. There is one major difference, however, and that is that Fortran has achieved a universality that is not possible for a machine-based language. Therefore, rather than dismiss it, let us examine its major weaknesses. It lacks (for our purposes):

(1) Important basic data types and data extensibility

(2) Good structure

(3) Dynamic storage allocation

However, these facilities are only missing in the syntactic structure of the language. Any data type can be coded as a subroutine and invoked by a call. Character springs or other multiple word data can be stored in arrays. Pointers can be represented as indices into arrays. Good syntactic representation of the required operations can be handled by a precompiler, as has been proposed by Cray (1973), Dunham (1971), Hull and Hofbauer (1974),

and many others. Such a precompiler can replace operation on, say, character springs, by calls to appropriate subroutines. Equally well, a precompiler can replace pointer operations by integer arithmetic on indices.

The organization of a program following good structural principles is possible. See Hull (1973), Woolley and Miller (1974), or Hochsprung (1974), for example, all of whom propose precompilers which convert a structured GOTO-less language into Fortran or PL/I. It is true that the resulting Fortran (or PL/I) code may not be as readable, and will offend the language purists - see Rosin (1974) - but if the precompiler is associated with proper debugging facilities, it is not necessary to refer to the Fortran program at any time.

Many large systems have implemented dynamic storage allocation by means of subroutines. In fact, any possible storage allocation scheme can be implemented in Fortran by assigning all of memory to one COMMON array and computing indices! Thus, the Algol and Fortran below are "equivalent".

Algol	Fortran
...	...
begin real A,B;	...
real array C[1:N];	STACK = STACK + ?
...	...
A:= C[2] + B;	Z(STACK) = Z(STACK+3)+Z(STACK+1)
begin real x;	STACK = STACK + 2 + N
...	...

Z is the memory array in Fortran COMMON and STACK is an allocation pointer. It is clear that any clarity of code is lost but the conversion could have been made by a precompiler so that it would not be necessary for the user to look at the Fortran. In fact, we can do almost anything we want in Fortran because it is almost as basic as an assembly language - and that, perhaps, is how we should view it.

A Proposal

Because Fortran is so basic and universal, I suggest we should declare it as the *machine language* for numerical programming. In the late '50s - early '60s there was a search for a universal compiler oriented language into which all compilers could translate, and via which all machine language code would be produced. Now that compiling and object code optimizing techniques have reached an advanced stage, Fortran fills that need nicely. Its basic operations are well matched to the capabilities of most machines, and what we need is a compiler - or set of compilers - that generate Fortran as object code. In this way, the object code of any compiler we use will be portable. If the preprocessor itself is written in one of these languages, it also will be portable. This means that the program author can ship "machine language" code on a portable basis to users who do not wish to change its code in any way, and he can also send a copy of the preprocessor to those users who wish to modify the code. Such a compiler is usually called a pre-processor, so we will use that name.

Perhaps the most important feature is that it gives us a medium in which to experiment with new language features. A pre-processor is a lot simpler than a compiler, and it is feasible for a numerical programmer to incorporate extensions to the language. The semantics of an extension can be defined by subroutines written in Fortran (or in the language recognized by the pre-processor). As experimental features achieve a level of acceptance, they can be incorporated into a standard pre-processor that is generally available. Code acceptable to that will be totally portable in source, but other code will always be object-portable, and portable in combination with its pre-processor. Even code written using features that are later discarded

will continue to be object-portable, so no code need be lost.

It is true that few pairs of machines will accept exactly the same version of Fortran - but that can be handled by pre-processor output options which determine the object language peculiarities. There is no need to stay with a minimum Fortran as object code for a particular installation. For example, 360 Fortran accepts index expressions. These certainly should be allowed by the precompiler, and no translation is needed when the object "machine" is 360 Fortran. However, for some other machines, it would be necessary to compile additional Fortran statements so that

```
B = B + A(I*J + K)
```

becomes

```
SYSVI = I*J + K
B = B + A(SYSVI)
```

where SYSVI is some variable distinct from anything currently in use. A very gereralized type of preprocessor can allow specification of these options by syntactic mapping descriptions between the input and output. Future changes in machines (that is, in their Fortran systems) can be met by changes in these options. Existing code need not be lost by such updates. Future improvements in object code optimizing (in the Fortran to machine language compiler) can be reflected in the efficiency of all code.

A second step that it is desirable to take is to reach agreement on a minimum input language for a precompiler so that experimental languages are derived from a common basis. Perhaps that minimum should be just Fortran, but I think not because it would place severe restrictions on any extension.

Needless to say, we have some ideas about what should be in the initial basic precompiler. We are developing one at Urbana. The preliminary version (written in 360 machine language by L. Lopez and B. VanMelle) is

PL/I-ish without GO TO's, much dynamic memory allocation, or recursion. The next version will be written in its own language and include better memory allocation. A short example of input and the output Fortran (!) is shown in figures 1 and 2. For various reasons to do with details I don't want to discuss here, we have translated all variables into an "internal" form including a $ sign. This gives a totally unreadable object code, but it is not our intention that it should be read! However, we do hope to find other solutions to the problems that lead to this, so that, as far as possible, the original names can be used in case the user wishes to change the object directly. Although the Fortran code it produces is considerably longer (in

```
END1:       DO          WHILE (IEX1 < 2);
              ...
              IF  DDS*R >= EPSN THEN DO;
                  DDS2 = DDS;
END2:             DO IEX2 = 0 TO 2;
                     CALL FUN(Y0,YD,DY,T,N);
                     CALL SOLVE(DY,DD1,D(1),N);
                     DDS = 0;
                     DDS1 = 0;
                     DO J = 1 TO N;
                        DDS1 = DDS1 + DD1(J)**2*ES(J);
                        Y0(J) = Y0(J) - DD1(J);
                        YD(J) = YD(J) - DD1(J)*D(1);
                        DD(J) = DD(J) + DD1(J);
                        DDS = DDS + DD(J)**2*ES(J);
                        END;
                     R2 = DDS1/DDS2;
                     DDS2 = DDS1;
                     IF R2>0.5D0 | DDS1>EPSN*10 THEN DO;
                        H = H/2;
                        LOOP END1;
                        END;
                     R1 = MAX(.9D0*R1,R2);
                     R = MIN(1,2*R1);
                     IF DDS1*R < EPSN THEN EXIT END2;
                     END END2;
                  END;
              IF DDS < EP THEN EXIT END1;
              H = H*(EP/DDS)**(0.5D0/K)*.9D0;
              END END1;
```

Figure 1. PL/W Input

```
      DO 0019 JA$003=1,1,I$ZERO
      IF(.NOT.(I$27.LT.2))GOTO 0020
      ...
      IF(.NOT.(D$54*D$41.GE.D$51))GOTO 0038
      D$56=D$54
      DO 0041 JA$005=1,3
      I$28=JA$005-1
      CALL MAIN00(D$34,D$35,D$36,D$45,I$21)
      CALL MAIN01(D$36,D$38,D$40(1),I$21)
      D$54=0
      D$55=0
      JC$006=I$21
      IF(1.GT.JC$006) GOTO 0044
      DO 0043 JA$006=1,JC$006
      I$20=JA$006
      D$55=D$55+D$38(I$20)**2*D$32(I$20)
      D$34(I$20)=D$34(I$20)-D$38(I$20)
      D$35(I$20)=D$35(I$20)-D$38(I$20)*D$40(1)
      D$37(I$20)=D$37(I$20)+D$38(I$20)
      D$54=D$54+D$37(I$20)**2*D$32(I$20)
0043  CONTINUE
0044  CONTINUE
      D$43=D$55/D$56
      D$56=D$55
      IF(.NOT.(D$43.GT.0.5D0.OR.D$55.GT.D$51*10))GOTO 0045
      D$47=D$47/2
      GOTO 0019
0045  CONTINUE
      D$42=DMAX1(.9D0*D$42,D$43)
      D$41=DMIN1(1D0,2*D$42)
      IF(.NOT.(D$55*D$41.LT.D$51))GOTO 0048
      GOTO 0042
0048  CONTINUE
0041  CONTINUE
0042  CONTINUE
0038  CONTINUE
      IF(.NOT.(D$54.LT.D$49))GOTO 0049
      GOTO 0020
0049  CONTINUE
      D$47=D$47*(D$49/D$54)**(0.5D0/I$17)*.9D0
0019  CONTINUE
0020  CONTINUE
```

Figure 2. Fortran Output

terms of number of statements) than Fortran code produced by hand, the final machine code is comparable in speed. The integration problem mentioned earlier was run through this pre-processor and then compiled by Fortran G and H. The results are shown in Table 3. (Although the result for the

Table 3. Speed Tests for PL/W

	C	D
H-level	1.89	.266
G-level	2.55	.768

H-level run was better than for direct coding using Fortran, there is a 10% variability in the times measured from one run to another because of multi-programming.)

In future versions we expect to allow for arbitrary programmer defined data types - the basic operations must be defined by programmer supplied subroutines. In particular, this will allow the variable precision operations proposed by Hull and Hofbauer (1974). After a new type is declared, operations (e.g. " + ") involving that type will generate calls to the programmer supplied subroutine.

Mechanisms for refinement will also be added. Hull and Hofbauer (1974) suggest one such mechanism. It is my feeling that a replacement should be limited to a syntactic entity, that is, it should be possible to write

```
IF    A > B
      THEN   [string 1]
      ELSE   [string 2]
```

where [string 1] and [string 2] are blocks, but it should not be possible to

write

IF A > B THEN [string 3]

where [string 3] is [string 1] ELSE [string 2] since the latter obscures the structure.

Although this type of refinement can be handled by a general macro facility (a preprocessor to the preprocessor!) it seems more advantageous to do it during the syntactic analysis of the input. Not only does refinement help the programmer build, debug and read the programmer; it can help the precompiler analyze it.

A fairly general pointer facility is allowed at present, but this is potentially dangerous as pointers can easily get out of hand. A form that restricts pointers so that they cannot point at anything that is invalid is desirable. See, for example, the class declaration in Dahl and Hoare (1972). (Unfortunately, this implies that some form of garbage collection must be done.)

We also plan to experiment with some type of macro pre-processing. One of the weaknesses of existing language compilers is that they are independent of their environments, particularly of the subroutine libraries. This was a desirable feature when compilers were slow and as little code as possible should be recompiled. Now it seems desirable to keep as much as possible of the user interface sections of code in source language so that it can be selectively modified.

One particular proposal we have concerns the provision of options in complex packages. There are two ways of providing these currently: different subroutines can be requested for each different option combination, or a set of numeric codes can be passed to a single subroutine. The first involves a decision at load time - it causes only the code needed to be

loaded, but the typical organization of such code as many interconnecting small subroutines leads to a large overhead of calls. The second decision is not made until run time so everything must be loaded, and dynamic tests made during execution to select the required code. These tests absorb time if they have to be made frequently. An alternative is to use a macro in the pre-processor. Subroutines could be stored in source and subject to a minimum pre-processing. In this pre-processing, those sections needed could be selected. Sections could be complete subroutines, as is possible with present loaders, or parts of them.

Selection could be made by conditional loading statements such as "IF ANY A IS SYMMETRIC THAN LOAD ... (source code) ... END", where A is a parameter used in calls of the package. Since a typical package will have many options, the concept of keyword parameters should be borrowed from macros. Instead of writing CALL SUB (V_1,V_2,...,V_N) where VI is the value of Ith of many parameters, we should be allowed to write CALL SUB(PI = VI, PJ = VJ) if only two parameters are to be given other than their default values. For example, CALL INTEGRATE(X,Y,FUN,N,START,EPS,ERROR=RELATIVE,STOP= 'Y(3) > 0') might be used to tailor a version of an integrator.

Similar features can be used to control program loading based on environment and execution parameters. For example, an "ASSERT <assertion>" statement could be compiled into IF <assertion> = FALSE THEN (print diagnostic) if a debug option was on, but ignored if it was off.

Conclusion

The objective of the type of system we are constructing is to build a flexible superstructure over existing systems so that we can continue

to use the vast amount of available code and yet not be caught in the confines of those old systems. Because of our earlier bad experiences we have totally embraced Fortran as a "machine language". All code will be written in it or preprocessed to it, and all interfaces between major sections that deal with "statements" will be in it or in a higher level form. Thus expressions for the value of, say, a resistor will be carried as Fortran expressions. In this way, we hope our packages will be truly portable and relatively easy for users to modify in those sections we feel they can modify safely.

Bibliography

Baecker, H. D., (1972) "On A Missing Mode in Algol 68", SIGPLAN Notices, Dec., 1972, pp. 20-30

Cody, W. J., (1974) "The Construction of Numerical Subroutine Libraries", SIAM Review, 16, Jan., 1974, pp. 36-46

Crary, F. D., (1973) "Language Extensions and Recompilers", Math. Research Center Technical Summary Report #1317, Feb., 1973, University of Wisconsin, Madison

Dahl, O. J., and Hoare, C. A. R. (1972) "Hierachical Program Structures" in "Structured Programming" by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Academic Press, London, 1972

Dijkstra, E. W., (1972) "Notes on Structured Programming" in "Structured Programming" by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Academic Press, London, 1972, pp. 1-82

Dunham, C. B. (1971) "Non-Standard Arithmetic" in Mathematical Software, ed. J. R. Rice, Academic Press, N.Y., 1971, pp. 105-111

Hoare, C. A. R. (1973) "Hints on Programming Language Design", Computer Science Department Report CS 403, Stanford University

Hochsprung, R. (1974) Letter to editor, Datamation, February, 1974, p. 18

Hull, T. E. (1973) "Would You Believe Structured Fortran?", SIGNUM Newsletter, October, 1973

Hull, T. E., Hofbauer, J. J. (1974) "Language Facilities for Numerical Computation", this conference

Rosin, R. F. (1974) Letter to editor, SIGNUM Newsletter, April, 1974, p. 22

Woolley, J. D., Miller, L. R. (1974) "LINUS: A Structured Language for Instructional Use", Proceedings SIGCSE Conference, Detroit, 1974, pp. 125-128

Wulf, W. A. (1972) "The Problem of the Definition of Subroutine Calling Conventions", SIGPLAN Notices, December, 1972, pp. 3-8

Form AEC–427
(6/68)
AECM 3201

U.S. ATOMIC ENERGY COMMISSION
UNIVERSITY–TYPE CONTRACTOR'S RECOMMENDATION FOR DISPOSITION OF SCIENTIFIC AND TECHNICAL DOCUMENT

(See Instructions on Reverse Side)

1. AEC REPORT NO.

COO-2383-0009

2. TITLE

What Do We Need In Programming Languages for Mathematical Software?

3. TYPE OF DOCUMENT (Check one):

- [X] a. Scientific and technical report
- [ ] b. Conference paper not to be published in a journal:
  - Title of conference _____
  - Date of conference _____
  - Exact location of conference _____
  - Sponsoring organization _____
- [ ] c. Other (Specify) _____

4. RECOMMENDED ANNOUNCEMENT AND DISTRIBUTION (Check one):

- [X] a. AEC's normal announcement and distribution procedures may be followed.
- [ ] b. Make available only within AEC and to AEC contractors and other U.S. Government agencies and their contractors.
- [ ] c. Make no announcement or distrubution.

5. REASON FOR RECOMMENDED RESTRICTIONS:

6. SUBMITTED BY: NAME AND POSITION (Please print or type)

C. W. Gear
Professor and Principal Investigator

Organization

University of Illinois
Department of Computer Science
Urbana, Illinois 61801

Signature

Date

May, 1974

FOR AEC USE ONLY

7. AEC CONTRACT ADMINISTRATOR'S COMMENTS, IF ANY, ON ABOVE ANNOUNCEMENT AND DISTRIBUTION RECOMMENDATION:

8. PATENT CLEARANCE:

- [ ] a. AEC patent clearance has been granted by responsible AEC patent group.
- [ ] b. Report has been sent to responsible AEC patent group for clearance.
- [ ] c. Patent clearance not required.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-74-652	2.	3. Recipient's Accession No.

4. Title and Subtitle

What Do We Need In Programming Languages for Mathematical Software?

5. Report Date

6.

7. Author(s)

C. W. Gear

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

University of Illinois

10. Project/Task/Work Unit No.

11. Contract/Grant No.

AEC AT(11-1)2383

12. Sponsoring Organization Name and Address

Atomic Energy Commission
Chicago Operations
9800 South Cass
Argonne, Illinois

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

Existing, widely used languages are reasonably satisfactory for writing numerical subroutines. However, future mathematical software development is likely to require features not currently found in any widely used language. Those features will be needed so that the program author can specify, at a meta-language level, the way in which his program will be tailored for a particular class of problems and particular hardware characteristics. It is suggested that the most fruitful way of doing this is via a preprocessor with some macro features that produces a widely used language such as Fortran as its object. Some specific suggestions are made based on our initial experience with one such preprocessor.

17. Key Words and Document Analysis. 17a. Descriptors

Numerical Software
Languages
Precompilers

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Unlimited	UNCLASSIFIED	26
	20. Security Class (This Page) UNCLASSIFIED	22. Price

JUL 23 1974

BOUND BY
HNM

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no.649-654(1974
Minimization of logic networks under e g
3 0112 088401309